The Linux Kernel Driver Model

Patrick Mochel
Open Source Development Lab
mochel@osdl.org

Abstract

A new device driver model has been designed and developed for the 2.5 version of the Linux Kernel. The original intent of this model was to provide a means to generically represent and operate on every device in a computer. This would allow modern power management routines to perform properly ordered suspend and resume operations on the entire device space during system power state transitions.

During the integration of the initial model, it became apparent that many more concepts related to devices besides power management information could be generalized and shared. The new driver model has evolved to provide generic representations of several objects, including:

- Devices
- Device drivers
- Bus drivers
- Device Classes
- Device Interfaces

The model also provides a set of functions to operate on each of those objects. This has provided the opportunity to consolidate and simplify a large quantity of kernel code and data structures.

The new driver model has been included in the mainstream kernel, since version 2.5.1, and continues to mature and evolve as more drivers are converted to exploit its infrastructure.

This document describes the design and implementation of the new driver model. It covers the description of the objects and their programming interfaces, as well as their conceptual and programmatic interactions with one another. It also includes appendices on the kobject infrastructure and the sysfs filesystem. These are two important features that were developed as a result of the driver model, yet were divorced from the model because of their more general purpose. They are covered in this document in the context of how they are used in the driver model.

1 Introduction

Theory

The driver model was created by analyzing the behavior of the PCI and USB bus drivers. These buses are ubiquitous in contemporary computer systems, and represent the majority of the devices that the kernel supports. They contain the most mature support for dynamic addition, removal, and power management of devices and drivers. These features had already been permeating other kernel bus drivers, with inspiration from the PCI and USB subsystems. They were considered appropriate bases of assumptions, and are described below.

These assumptions hold true for the majority of instances of objects represented by the driver model. The minority of objects are considered exceptions, and must work around the driver model assumptions.

A bus driver is a set of code that communicates with a peripheral bus of a certain type. Some examples of bus drivers in the kernel include:

- PCI
- USB
- SCSI
- IDE
- PCMCIA

Bus drivers maintain a list of devices that are present on all instances of that bus type, and a list of registered drivers. A bus driver is compile-time option, and may usually be compiled as a module. There will never be multiple instances of a driver for the same bus type at the same time. Most, if not all, of its internal data structures are statically allocated.

A bus instance represents the presence of a bus of a particular type, such as PCI Bus 0. They are dynamically allocated when the bus instance is discovered. They contain a list of devices, as well as data describing the connection to the device. Bus instances are not currently represented in the driver model. However, they are mentioned here to note the distinction between them and bus driver objects.

During startup, a bus driver discovers instances of its bus type. It scans the bus for present devices and allocates data structures to describe each device. Most of this information is bus-specific, such as the bus address of the device and device identification information. The bus driver need not know of the function the device performs. This object is inserted into the bus's list of devices, and the bus driver attempts to bind it to a device driver.

A device class describes a function that a device performs, regardless of the bus on which a particular device resides. Examples of device classes are:

- Audio output devices
- Network devices

- Disks
- Input devices

A device class driver maintains lists of device and drivers that belong to that class. They are optional bodies of code that may be compiled as a module. Its data structures are statically allocated. A device class defines object types to describe the devices registered wit h it.These objects define the devices in the context of the class only, since classes are independent of a device's bus type.

A device class is characterized by a set of interfaces that allow user processes to communicate with devices of their type. An interface defines a protocol for communicating with those devices, which is characterized in a user space device node. A class may contain multiple interfaces for communicating with devices, though the devices and drivers of that class may not support all the interfaces of the class.

A device driver is a body of code that implements one or more interfaces of a device class for a set of devices on a specific bus type. They contain some statically allocated structures to describe the driver to its bus and class, and to maintain a list of bound devices.

During startup, a driver registers with its bus, and the bus inserts the driver into its internal list of drivers. The bus then attempts to bind the driver to the bus's list of devices. A bus compares a list of bus-specific device ID numbers that it supports with the ID numbers of the devices on the bus. If a match is made, the device is 'attached' to the driver. The driver allocates a driver-specific object to describe the device. This usually includes a class-specific object as well, which the driver uses to register the device with it's class.

Infrastructure

The driver model provides a set of objects to represent the entities described above. Previously, the kernel representations of these objects, especially devices, have varied greatly based on the functionality that they provide and the bus type forwhich they are designed. Much of this information is specific to the type of object, though some can be generically encoded, which are what compose the driver model objects.

Most of the members of driver model objects contian meta data describing the object, and linkage information to represent membership in groups and lists of subordinate objects. The driver model object definitions can be found in the header file include/linux/device.h.

The driver model also provides a set of functions to operate on each object. These functions are referred to the 'driver model core' through this document. They are implemented in drivers/base/ in the kernel source tree. Each set of operations is object-specific and described in detail with the objects in their respective sections. However, there are some commonalities between objects that are reflected in the functions of the core. These are described here to provide perspective on the model, and to avoid repeating the purposes in each section.

Every driver model object supports dynamic addition and removal. Each object has register() and unregister() functions specific to that object. Registration initializes the object, inserts it into a list of similar objects, and creates a directory for the object in the sysfs filesystem. Registration occurs when a device is discovered or a module containing a device, bus, or class driver is inserted. Unregistration occurs when a driver's module is unloaded or a device is physically removed. Unregistration removes the object's sysfs directory and deletes it from the list of similar objects.

To support dynamic removal, each object contains a reference count that can be adjusted by object-specific get() and put() functions. The get() routine should be called for an object before it is used in a function. get() returns a reference to the object. The put() routine should be called after a function is done using an object. It has no return value.

If, and only if, an object's reference count reaches 0 may the object be freed, or its module be unloaded. Another process may still hold a reference to the object when the object's unregister() function returns. Blindly freeing an object's memory could cause the other process to access invalid or reallocated memory. The method that the driver model uses to prevent these bugs is described in the individual objects' sections.

The similarities of driver model objects, and the amount of replicated code to perform similar operations on the objects prompted the effort to create a generic object type to be shared across not only driver model objects, butalso across all complex kernel objects that supported dynamic registration and reference counting.

A kobject generically describes an object. It may be embedded in larger data types to provide generic object meta data and a reference counting mechanism. A subsystem represents a set of kobjects. A kobject can be dynamically registered and unregistered with their subsystem. When a kobject is registered, it is initialized and inserted into its subsystem's list of objects. A directory in sysfs, described below, is also created. Unregistration removes the sysfs directory for the kobject and deletes it from its subsystem's list. Kobjects also have a reference count and get() and put() operations to adjust it. A kobject may belong to only one subsystem, and a subsystem must contain only identically embedded kobjects.

The kobject functionality is very similar to the driver model core. The driver model functions have been adapted to use the kobject infrastructure, and some provide no more functionality than calling the associated kobject function.

The kobject and subsystem infrastructure is described in Appendix A.

The sysfs filesystem was originally created as a means to export driver model objects and their attributes to user space. It has since been integrated with the kobject infrastructure, so it can communicate directly with the generic object, allowing it to do reference counting on represented objects.

All registered kobjects receive a directory in sysfs in a specific location based on the kobject's ancestry. This provides user space with a meaningful object hierarchy in the filesystem which can be navigated through using basic file system tools, like ls(1), find(1), and grep(1).

Attributes of an object may be exported via sysfs, and are represented as text files in the filesystem. Sysfs provides a means for user processes to read and write attributes. The kernel exporters of attributes may define methods called by sysfs when a read or write operation is performed on an attribute file.

The sysfs filesystem and its programming interface is discussed in Appendix B.

Summary

The new driver model implements a large amount of infrastructure for describing and operating on device-related objects. This infrastructure has been able to eliminate redundant kernel code and data structures, especially in the areas of object reference counting and list membership management. This reduces the complexity of device, bus and class drivers, making them easier to write and easier to read. The consolidation of data structures also makes it easier to implement common functions to operate on objects of traditionally different objects.

While this document describes the base objects and operations of the driver model, there are many things it omits. It does not describe how it is exploited for the purpose of power management or implementing a dynamic and scalable device naming scheme. These topics are beyond the scope of this document and reserved for other discussions.

Similarly, this document only superficially describes the kobject infrastructure and the sysfs filesystem. These are included only to provide enough context to complete the discussion of the driver model, and are best covered in other documents.

2 kobjects and subsystems

The kobject infrastructure was developed to consolidate common aspects of driver model objects. The result is a simple object type designed to provide a foundation for more complex object types. Struct subsystem was created to describe and manage sets of kobjects. It was also created to consolidate and unify driver model objects and functions.

kobjects

Struct kobject provides basic object attributes in a structure that is designed to be embedded in larger structures. Subsystems that embed kobject may useits members, and the helpers for them, rather than defining their own.

When a kobject is registered, it should initialize the following fields:

- name
- parent

Name	*Type*	*Description*
name	char [KOBJ_NAME_LEN]	Name of object.
refcount	atomic_t	Reference count of kobject.
entry	struct list_head	List entry in subsystem's list of objects.
parent	struct kobject *	Parent object.
subsys	struct subsystem *	kobject's subsystem.
dentry	struct dentry *	Dentry of object's sysfs directory.

Table 1: struct kobject data fields

- subsys

'name' gives the object identity, and provides a name for its sysfs directory that is created. 'parent' provides ancestral context for the object. The kobject infrastructure inserts the kobject into an ordered list that resides in its subsystem. The ordering is dependent on the kobject's parent - the object is inserted directly before its parent. This ordering guarantees a depth-first ordering when iterating over the list forwards, and a breadth first ordering when iterating over it backwards.

The 'subsys' informs the kobject core of the kobject's controlling subsystem. A kobject may belong to only one subsystem at a time. Note that a kobject's parent is set to point to its subsystem, if its parent pointer is not set by the kobject core.

kobject Programming Interface

Kobjects may be dynamically added and removed from their subsystem. The kobject programming interface provides two parallelprogramming models, much like devices do. kobject_register() initializes the kobject, and adds to the kobject hierarchy. The same action may also be performed by calling kobject_init() and kobject_add() manually.

kobject_unregister() removes the kobject from the system and decrements its reference count. When the reference count of the kobject reaches 0, kobject_cleanup() will be called to tear down the kobject. Alternatively, one may call kobject_del() and kobject_put() to obtain the same results. kobject_cleanup() is called only by kobject_put(), and should never be called manually.

The parallel interface is provided to afford users of the kobject model more flexibility. kobject_add() inserts the kobject into its subsystem and creates a sysfs directory for it. A user may not desire that to happen immediately, or at all. kobject_init() may be called to initialize the kobject to a state in which the reference count is usable. At a later time, the user may call kobject_add() add the device to the subsystem and create its sysfs directory.

Users of the low-level initialize and add calls should also use the low-level kobject_del() and kobject_put() calls, even if they happen consecutively, like kobject_unregister(). Even though kobject_register() and kobject_unregister()

Name	*Return Type*	*Description*
kobject_init(struct kobject *)	void	Initialize kobject only.
kobject_cleanup(struct kobject *)	void	Tear down kobject, by calling its subsystem's release() method. Used internally only.
kobject_add(struct kobject *)	int	Add kobject to object hierarchy.
kobject_del(struct kobject *)	void	Remove kobject from object hierarchy.
kobject_register(struct kobject *)	int	Initialize kobject AND add to object hierarchy.
kobject_unregister(struct kobject *)	void	Remove kobject from object hierarchy and decrement reference count.
kobject_get(struct kobject *)	struct kobject *	Increment reference count of kobject.
kobject_put(struct kobject *)	void	Decrement reference count of kobject, and tear it down if the count reaches 0.

Table 2: kobject Programming Interface

currently do no extra work, they are not excluded from ever doing so. Adhering to the symmetry also makes the code easier to follow and understand.

A kobject contains a reference count that should be incremented – using kobject_get() – before accessing it, and decremented - using kobject_put() – after it's not being used any more. Keeping a positive reference count on an object guarantees that the structure will not be removed and freed while it's being used.

Subsystems

struct subsystem was defined to describe a collection of kobjects. 'subsystem' is an ambiguous name for such a generic object; 'container' is more accurate, and will likely change at a later date. The object contains members to manage its set of subordinate kobjects. Users of subsystem objects may embed the structure in more complex objects and use the fields contained in it, rather than defining their own.

Struct subsystem contains an embedded kobject to contain generic objects meta data about the subsystem itself. This is also to represent the subsystem's membership in another subsystem.

The subsystem object contains a list that registered objects are inserted on. 'rwsem' is a semaphore that protects access to the list. It should always b e taken before iterating over the list, or adding or removing members.

Subsystems are believed to be hierarchical, as some subsystems contain subordinate subsystems or containers of objects. The 'parent' field may be used to denote a subsystem's ancestral parent.

Name	*Type*	*Description*
kobj	struct kobject	Generic object metadata.
list	struct list_head	List of registered objects.
rwsem	struct rw_semaphore	Read/write semaphore for subsystem.
parent	struct subsystem *	Parent subsystem.
sysfs_ops	struct sysfs_ops *	Operations for reading and writing subordinate kobject attributes via sysfs.
default_attrs	struct attribute **	NULL-terminated array of attributes exported via sysfs for every kobject registered with subsystem.

Table 3: struct subsystem data fields

Name	*Return Type*	*Description*
release(struct device *)	void	Method called when a registered kobject's reference count reaches 0. Used by the subsystem to free memory allocated for kobject.

Table 4: struct subsystem methods.

'sysfs_ops' is a pointer to a set of operations that a subsystem must define to read and write attributes that are exported for subordinate kobjects via the sysfs filesystem. 'default_attrs' is a NULL-terminated array of attributes that are unconditionally exported for every kobject registered with the subsystem. sysfs is discussed, and these fields will be covered, in the next section.

The 'release' method is defined as a means for the subsystem to tear down a kobject registered with it. A subsystem should implement this method. When a kobject's reference count reaches 0, kobject_cleanup() is called to tear down the device. That reference's the kobject's subsystem and its 'release' method, which it calls to allow the subsystem to tear down the device (Since the subsystem likely embedded the kobject in something larger, and it must convert to it.).

Programming Interface

Subsystems provide a similar programming model to kobjects. However, they allow only simple register() and unregister() semantics, and do not export the intermediate calls for the rest of the kernel to use. Also , a subsystem's reference count may be incremented using subsys_get() and decremented using subsys_put().

Name	*Return Type*	*Description*
subsystem_register(struct subsystem *)	int	Initialize subsystem and register embedded kobject.
subsystem_unregister(struct subsystem *)	void	Unregister embedded kobject.
subsys_get(struct subsystem *)	struct subsystem *	Increment subsystem's reference count.
subsys_put(struct subsystem *)	void	Decrement subsystem's reference count.

Table 5: struct subsystem Programming Interface.

3 The sysfs filesystem

sysfs is an in-memory filesystem with a kernel programming interface for exporting object and their attributes. The purpose of sysfs is to export kernel objects and their attributes. Sysfs is directly related to the kobject infrastructure; every kobject that is added to the system has a directory created for it in sysfs. Every directory that is created in sysfs must have a kobject associated with it.

Sysfs uses the kobject hierarchy information to determine where to create an object's directory. This enables sysfs to expose object hierarchies, like the device hierarchy, with no additional overhead. This also preventsa chaotic directory structure, which is characteristic of the procfs filesystem. The following output is from the tree(1) command, and expresses the device hierarchy, as represented in sysfs.

```
# tree -d /sys/devices/
/sys/devices/
|-- ide0
|   |-- 0.0
|   `-- 0.1
|-- ide1
|   |-- 1.0
|   `-- 1.1
|-- legacy
|-- pci0
|   |-- 00:00.0
|   |-- 00:01.0
|   |   `-- 01:05.0
|   |-- 00:07.0
|   |-- 00:07.1
|   |-- 00:07.3
|   |-- 00:07.4
|   |-- 00:09.0
|   |-- 00:09.1
|   |-- 00:09.2
|   |-- 00:0b.0
|   `-- 00:0c.0
`-- sys
    |-- cpu0
```

Name	*Return Type*	*Description*
sysfs_create_dir(struct kobject *)	int	Create sysfs directory for kobject.
sysfs_remove_dir(struct kobject *)	void	Remove sysfs directory of kobject.

Table 6: sysfs Programming Interface.

```
|-- cpu1
|-- pic0
`-- rtc0
```

Sysfs provides an extensibleinterface for exporting attributes of objects. Attributes are represented by regular files in a kobject's directory, with a preference for the files to be in ASCII text format with one value per file. These preferences are not encoded programmatically, butobjects exporting attributes are strongly encouraged to follow this model. The following is output. The following shows thecontents of a PCI device's directory, which contains four attribute files:

```
# tree /sys/devices/pci0/00:07.0/
/sys/devices/pci0/00:07.0/
|-- irq
|-- name
|-- power
`-- resource

0 directories, 4 files
```

sysfs has been a standard part of the Linux kernel as of version 2.5.45. It has existed under a different name (driverfs or ddfs) since kernel version 2.5.2. The de-facto standard mount point for sysfs is a new directory named '/sys'. It may be mounted from user pace by doing:

```
# mount -t sysfs sysfs /sys
```

Basic Operation

The sysfs header file is include/linux/sysfs.h. It contains the definition of struct attribute and the prototypes for managing object directories and attributes. Sysfs directories are created for kobjects when the kobjects are added to the kobject hierarchy. The directories are removed when the kobjects are deleted from the hierarchy.

Name	*Type*	*Description*
name	char *	Name of attribute.
mode	mode_t	Filesystem permissions for attribute file.

Table 7: struct attribute Data Fields.

Name	*Return Type*	*Description*
sysfs_create_file(struct kobject *, struct attribute *)	int	Create attribute file for kobject.
sysfs_remove_file(struct kobject *, struct attribute *)	void	Remove attribute file of kobject.

Table 8: struct attribute Programming Interface.

Attributes

In their simplest form, sysfs attributes consist of only a name and the permissions for accessing the attribute. They provide no details about the type of data presented by the attribute, or the object exporting the attribute. The means for reading and writing attributes are subsystem-dependent, and are discussed next.

An attribute may be added to the kobject's directory by calling sysfs_create_file(). The function references the attribute structure, and creates a file named 'name' with a mode of 'mode' in the kobject's directory. The attribute can be removed by calling sysfs_remove_file(), passing pointers to the kobject and the attribute descriptor.

It is important to note that attributes do not have to be specific to one instance of an object. A single attribute descriptor may be reused for any number of object instances. For instance, a single attribute descriptor exports a device's name. An attribute file is created for each device that is registered with the system, though the same attribute descriptor is passed to sysfs_create_file() for each one.

Reading and Writing Attributes

Reading and writing kobject attributes involves passing data between the user space process that is accessing an attribute file and the entity that has exported the attribute. The base definition of attributes does not provide this type of functionality. The definition is ignorant of the type of the attribute and the specific data it represents. In order to get this data, sysfs must call one of the methods defined by the subsystem the kobject belongs to in its struct sysfs_ops object.

Name	*Return Type*	*Description*
show(struct kobject *, struct attribute *, char *, size_t, loff_t)	ssize_t	Called by sysfs when a user space process is reading an attribute file. Data may be returned in the PAGE_SIZE buffer.
store(struct kobject *, struct attribute *, const char *, size_t, loff_t)	ssize_t	Called by sysfs when a user space process writes to an attribute file. Data is stored in the PAGE_SIZE buffer passed in.

Table 9: struct attribute I/O methods.

Sysfs calls a subsystem's show method when a user mode process performs a read(2) operation on an attribute. It calls a subsystem's show method when a write(2) is performed on the file. In both cases, the size of the buffer passed to the methods is the size of one page.

The current model places all responsibility of handling the data on the downstream functions. They must format the data, handle partial reads, and correctly handle seeks or consecutive reads. In this case, sysfs is much likethe default procfs operation, without using the seq_file interface. Though sysfs allows file creation and the downstream functions to be much simpler, the interface is still considered prohibitivelycomplex. A new interface is being developed that will ease the burden of the downstream formatting and parsing functions.

Extending Attributes

Attributes may be added for an object at any time, by any type of kernel entity, whether they are a core part of the subsystem, a dynamically loaded driver for the kobject, or even some proprietary module. There is only one set of sysfs_ops for a subsystem, though, so struct attribute may be extended to describe the secondary methods necessary to read or write an attribute.

Extending an attribute can be done for a type of object by defining a new type of attribute, that contain an embedded struct attribute and methods to read and write the data of the attribute. For example, the following definitions exist in include/linux/device.h.

```
struct device_attribute {
        struct attribute        attr;
        ssize_t (*show)(struct device *, char *, size_t, loff_t);
        ssize_t (*store)(struct device *, const char *, size_t, loff_t);
};

int device_create_file(struct device *, struct device_attribute *);
```

```
void device_remove_file(struct device *, struct device_attribute *);
```

Kernel components must define an object of this type to export an attribute of a device. The device_attribute object is known by the device subsystem. When it receives a call from sysfs to read or write an attribute, it converts the attribute to a device_attribute object, as well as converting the kobject to a struct device.

Components that export attributes for devices may define 'show' and 'store' methods to read and write that attribute. An explicit device object is passed to these methods, so the definer of the method does not have to manually convert the object. The methods are also not passed a pointer to the attribute, as it should be implicit in the method being called.

Reference Counting

Composite in-kernel filesystems suffer from potential race conditions when user space is accessing objects exported by them. The object could be removed, either by removing a physical device or by unloading a module, while a user space process is still expecting a valid object. Sysfs attempts to avoid this by integrating the kobject semantics into the core of the filesystem. Note that this is only an attempt. No current race conditions exist, though their existence is not precluded entirely.

When a directory is created for a kobject,a pointer to the kobject is stored in the d_fsdata field of the dentry of the directory. When an attribute file is created, a pointer to the attribute is stored in the d_fsdata field of the file. When an attribute file is opened, the reference count on the kobject is incremented. When the file is closed, the reference count of the kobject is decremented. This prevents the kobject, and the structure its embedded in, from being removed while the file is open.

Note that this does not prevent a device from being physically removed, or a module being unloaded. Downstream calls should always verify that the object a kobject refers to is still valid, and not rely on the presence of the structure to determine that.

Expressing Object Relationships

Objects throughout the kernel are referenced by multiple subsystems. In many cases, each subsystem has a different object to describe an entity, relevant to its own local context. Displaying only the objects in relation to the subsystem is an incredibly handy feature of sysfs. But, being able to symbolically and graphically represent the relationship between objects of two different subsystems can

Name	*Return Type*	*Description*
sysfs_create_link(struct kobject * kobj, struct kobject * target, char * name)	int	Create symbolic link in kobj's sysfs directory that points to target's directory, with a name of 'name'.
sysfs_remove_link(struct kobject *, char * name)	void	Remove symbolic link from kobject's directory with specified name.

Table 10: sysfs Symbolic Link Interface.

be invaluable. Sysfs provides a very simple means to do so, via symbolic links in the filesystem.

An example use of symbolic links is in the block subsystem . When a block device is registered, a symbolic link is created to the device's directory in the physical hierarchy. A symbolic link is also created in the device's directory that points to the corresponding directory under the block directory.

```
# tree -d /sys/block/hda/
/sys/block/hda/
|-- device -> ../../devices/ide0/0.0
|-- hda1
|-- hda2
|-- hda3
|-- hda4
|-- hda5
`-- hda6

7 directories
# tree -d /sys/devices/ide0/0.0/
/sys/devices/ide0/0.0/
`-- block -> ../../../block/hda

1 directory
```

4 Hotplug

The driver model supports dynamic addition and removal of all object types through their register() and unregister() functions. This includes support for the dynamic addition and removal of devices on bus types that electrically support it. This commonly referred to as the 'hotplug' capability or feature.

When a device's physical or logical status changes, the driver model executes the user mode program in /proc/sys/kernel/hotplug. This isset to '/sbin/hotplug' by default. This user mode agent may invoke user-defined policy for each action or each device, including:

- Loading or unloading a driver module.

Name	*Return Type*	*Description*
dev_hotplug(struct device *dev, const char *action)	int	Called by driver model core when device is registered or unregistered with core.
class_hotplug(struct device *dev, const char *action)	int	Called by driver model core when device is registered or unregistered with class.

Table 11: struct subsystem Programming Interface.

- Configuring a device.
- Executing a script or application.

The driver model calls one of two internal functions, selected by the source of the event.

The action for both functions is one of "ADD" or "REMOVE." These actions are set in environment variables for the the hotplug program. The driver model sets the following parameters:

- HOME, set to '/'
- PATH, set to '/sbin:/bin:/usr/sbin:/usr/bin'
- ACTION, set to 'ADD' or 'REMOVE'.
- DEVPATH, set to the sysfs path to the device's directory.

Additionally, the driver model defines a 'hotplug' methodforstruct bus_type and struct device_class.

```
int(*hotplug) (struct device *dev, char **envp,
               int num_envp, char *buffer, int buffer_size);
```

This method is called immediately before the user mode program is executed. If this method is defined, the bus or class driver may supply additional environment variables that bus or class specific policy may use in user space.

5 Devices

A device is a physical component of a computer that performs a discrete function that the kernel can exert some control over. A device contains the electrical components to perform its specified function, also known as its device class. A device resides on abus of a certain type. A bus defines a standard architectures for devices that reside on it, so devices of any function exist on it.

The driver model defines struct device to describe a device independent of the bus it resides on, or the function it performs. This generic description contains few members related directly to physical attributes of the device. Itinstead is a

Name	*Return Type*	*Description*
bus_list	struct list_head	List entry in bus's list of devices.
driver_list	struct list_head	List entry in driver's list of devices.
intf_list	struct list_head	List of interface descriptors for the device.
parent	struct device	Pointer to parent device.
kobj	struct kobject	Generic kernel object metadata.
name	char[DEVICE_NAME_SIZE]	ASCII description of device.
bus_id	char[BUS_ID_SIZE]	ASCII representation of device's bus-specific address.
bus	struct bus_type *	Pointer to bus type device belongs to.
driver	struct device_driver *	Pointer to device's controlling driver.
driver_data	void *	Device-specific data private to driver.
class_num	u32	Enumerated number of device within its class.
class_data	void *	Device-specific data private to class.
platform_data	void *	Device-specific data private to the platform.
power_state	u32	Current power state of the device.
saved_state	void *	Pointer to saved state for device.
dma_mask	dma_mask_t *	DMA address mask the device can support.

Table 12: struct device Data Fields.

Name	*Return Type*	*Description*
release(struct device *)	void	Garbage collection method for tearing down devices.

Table 13: struct device Method.

means to consolidate the similar aspects of the disparate device representations into a common place.

Struct device members fall into three basic categories - meta data, linkage information, and physical attributes. A device's meta data consists of fields to describe the device andits reference count.

A device's embedded kobject contains the reference count of the device and a short 'name' field. Kobjects are described in detail in Appendix A. Additionally, struct device contains a 'name' field of its own and a 'bus_id' field. The latter serves the same purpose as the kobject's name field - to provide an ASCII identifier unique only in a local context, i.e. among siblings of a common parent. This redundancy is known, and will be resolved in the future.

A device's 'name' field provides a longer string which can be used to provide a user-friendly description of a device, such as "ATI Technologies Inc Radeon QD" This description is provided by the bus driver when a device is discovered by looking the device's ID up in a table of names.

Struct device also contains several pointers to describe the objects that the device is associated with. Many of these fields are used by the core when registering and unregistering a device to determine the objects to notify. The 'bus' field should be set by the bus driver before registering a device. The core uses this to determine which bus to add it to. The 'driver' and field is set by the core when a device is bound to a driver. The 'class_num' field is set once the device is registered with and enumerated by the class its driver belongs to.

The data fields are used so subsystems can share data about an object without having to setup auxillary means of associating a device structure with a subsystem-specific object. 'driver_data' may be used by the driver to store a driver-specific object for the device. 'class_data' may be used by the class to store a class-specific object for the device. This should be set during the class's add_device() callback, as that is when the object is usually allocated. This object may also be used by a class's interfaces, since they are conscience of the class's internal representations.

'platform_data' is a pointer that the platform driver can use to store platform-specific data about the device. For example, in ACPI-based system, the firmware enumerates and stores data about many devices in the system. The ACPI driver can store a pointer to this data in 'platform_data' for later use.

This feature was an early requirement by ACPI. It has not been used since its addition, nor has it proven useful to other platforms besides ACPI-enabled ones. Therefore, it may be removed in the near future.

A device's linkage information is represented in the several list entry structures in the object. Itbecomes a member of several groups when it is registered with various drivers. The embedded kobject also contains a list entry representing membership in the global device subsystem. A device also has a pointer to its parent device, which is used by the core to determine some a device's ancestry.

There are few physical attribute s contained in struct device, since the number of physical attributes common to a majority of devices are very few in number. 'power_state' is the current power state of the device. Many bus types

define multiple device power states. PCI defines four power states, D0-D3 for PCI devices, and ACPI modeled their generic device power state definition after this. The D0 state is defined as being on and running, and the D3 state is defined as being off. All devices support these states by default. D1 and D2 are intermediate power states are usually optional for devices to support, even if the underlying bus can support these states. In these states, the device is unusable, but not all of the components of the device have been powered down. This reduces the latency of restoring the device to the usable D0 state.

When a device is suspended, physical components in the device are turned off to save power. When these components are turned back on, they are supposed to be restored to the identical state they were in before they were suspended. 'saved_state' is a field that the device driver can use to store device context when the device is placed in a low power state.

Both 'power_state' and 'saved_state' are fields that were added to struct device when the sole motivation was implementing sane power management infrastructure. Not all devices support power management beyond being 'On' or 'Off'. Many systems also do not support global power management, and many configurations do not support any type of power management mechanism. Because of this, these fields are being considered for removal. In order to do, though, some other mechanism for attaching power state information to the device must be derived. Note that such a mechanism could be used to attach platform-specific data to a device only when the system and the device support it.

'dma_mask' describes the DMA address mask that the device can support. This is obviously only relevant if the device supports DMA-able I/O. Several bus structures contain a dma_mask, which motivated the transition of it to the generic struct device. It is to be set by the bus driver when the device is discovered. Currently, dma_mask is only a pointer, which should point to the field in the bus-specific structure. This should be set by the bus driver before the device is registered. In the future, the field will be moved entirely to the generic device.

The 'release' method acts as a desctructor for the device object. It is called when a device's reference count reaches 0 to inform the object that allocated it that it is safe to free the device. Since struct device is usually embedded in a larger object, the larger object should be freed.

This method should be initialized by the object that discovers and allocates the device. This is typically the device's bus driver. Since there should be only one type ofdevice object that a bus controls, this method may bemoved to the struct bus_type object in the future.

Programming Interface

The struct device programming interface consists of two parallel models. One model is the basic registration and reference counting model common throughout the driver model. However, the interface exposes the intermediate calls that

Name	*Return Type*	*Description*
device_register(struct device * dev)	int	Initialize device and add it to hierarchy.
device_unregister(struct device * dev)	void	Remove device from hierarchy and decrement reference count.
device_initialize(struct device * dev)	void	Initialize device structure.
device_add(struct device * dev)	int	Add device to device hierarchy.
device_del(struct device * dev)	void	Remove device from hierarchy.
get_device(struct device * dev)	struct device *	Increment reference count of device.
put_device(struct device * dev)	void	Decrement device reference count.

Table 14: struct device Programming Interface.

device_register() and device_unregister() use to express a finer level of control over the lifetime of a device. The functions are described in the table below.

device_register() calls device_initialize() and device_add() to initialize a device and insert it into the device hierarchy respectively. Calling device_register() is the preferred means of registering a device, though the intermediate calls may be used instead to achieve the same result.

Analogously, device_unregister() calls device_del() and put_device() to remove the device from the device hierarchy and decrement its reference count. These intermediate calls may also be used to achieve the same result.

The intermediate interface is exposed to let bus drivers reference and use device objects before advertising their presence via the hierarchy and sysfs. This is necessary for USB devices and hot-pluggable bus types. To initialize a device after it is discovered, a series of I/O transfers must take place. The device may be removed during this process, so the subsystem must be aware of its reference count and potential garbage collection. However, the device is not fully functional yet, and must not be registered in the hierarchy.

6 Buses

A bus driver is a set of code that communicates with a peripheral bus of a certain type. Some examples of bus drivers in the kernel include:

- PCI
- USB
- SCSI
- IDE
- PCMCIA

Name	*Type*	*Description*
name	char *	The name of the bus.
subsys	struct subsystem	Bus's collection of subordinate objects.
drvsubsys	struct subsystem	Bus's collection of subordinate drivers.
devsubsys	struct subsystem	Bus' collection of subordinate devices.
devices	struct list_head	Bus's list of devices registered with it.

Table 15: struct bus_type Data Fields.

Name	*Type*	*Description*
match(struct device * dev, struct device_driver * drv)	int	Called during driver binding process for bus to compare a device's ID, with the list of Ids that the driver supports.
add(struct device * parent, char * bus_id)	struct device *	Called to add a device to a bus at a certain location.
hotplug(struct device *dev, char **envp, int num_envp, char *buffer, int buffer_size)	int	Called before /sbin/hotplgu is called on device insertion or removal, so bus can format parameters correctly.

Table 16: struct bus_type Methods.

Bus drivers maintain a list of devices that are present on the bus and a list of drivers that have registered with it. A bus driver is an optional compile option, and usually may be compiled as a module to be loaded at runtime. It's existence is singular. There will never be multiple instances of a driver for the same bus type at the same time. Therefore most, if not all, of its internal data structures are statically allocated.

A bus instance represents the presence of a bus of a particular type, such as PCI Bus 0. They are dynamically allocated when the bus instance is discovered. They contain a list of subordinate devices, as well as data describing the physical device the bus is connected to (the bridge) and membership information about which bus type it belongs to. Bus instances are not currently represented in the driver model. However, they are mentioned here to note the distinction between them and bus driver objects.

The driver model defines struct bus_type to represent instances of bus drivers, and is described below. The driver model does not define an object to describe a bus instance yet. Thisfeature may be added in the future.

The members of struct bus_type center around the lists of objects that buses maintain. bus_type contains three subordinate subsystems and a struct list_head to manage subordinate objects, though not all are fully used.

'subsys' is reserved for managing the set of bus instances ofa type, and is not currently used for that purpose. It is registered with the kobject core, though,

as a child of the global bus subsystem. This gives the bus a node in the kobject hierarchy , and a directory in the sysfs filesystem.

'drvsubsys' manages the set of drivers registered with the bus type. It is currently used, and the drivers registered with the bus are inserted into the subsystem's list.

'devsubsys' is intended to manage the list of devices present on all bus instances of the bus type. However, the bus must use another list tocontain the device objects. Kobjects may only belong to one subsystem, and the embedded kobject in struct device is a member of the global device hierarchy. Therefore, struct device contains a list entry 'bus_list' for insertion into the bus's list of devices.

Note that the list of devices that the bus maintains is the list of all devices on all buses of that type in the system. For example, a system may have several PCI buses, including multiple Host PCI buses. Though devices may be not be on the same physical bus, they all belong to the same bus type. This aides in routines that must access all devices of a bus type, like when binding a driver to devices.

The 'name' field of the bus is the name of the bus. This field is redundant with the name field of the embedded subsystem's kobject. This field may be removed in the future.

The methods of struct bus_type are helpers that the core can call to perform bus-specific actions too specific for the core to efficiently handle.

The 'match' method is used during the process of attaching devices to drivers. Devices drivers typically maintain a table of device IDs that they supports, and devices each contain an ID. These IDs are bus-specific, as are the semantics for comparing them. The driver model core can perform most of the actions of driver binding, such as walking the list of drivers when a device is inserted, or walking the list of devices when a driver module is loaded. But, it does not know what format the device ID table is in, or how it should compare them (i.e. if it should take into account any masks in the device IDs). The core instead calls the bus's match method to check if a driver supports a particular device.

The 'add' callback is available for other kernel objects to tell the bus driver to look for a particular device at a particular location. Other objects may know the format of a device's ID, but only the bus driver knows how to communicate with the deviceover the bus.

This is useful for firmware drivers that can obtain a list of devices from the firmware, and wish to tell the kernel about them, instead of having the kernel probe for them. So far, this method is not implemented by any bus drivers, or used by any code in the kernel. This method is not being considered for removal, though, since its neglect is due to a lack of time rather than a lack of usefulness.

The 'hotplug' method is called by the driver model core before executing the user mode hotplug agent. The bus is allowed to specify additional environment variables when the agent is executed.

Name	*Return Type*	*Description*
bus_register(struct bus_type * bus)	int	Register bus driver with driver model core.
bus_unregister(struct bus_type * bus)	void	Unregister bus driver from driver model core.
get_bus(struct bus_type * bus)	struct bus_type *	Increment reference count of bus driver.
put_bus(struct bus_type * bus)	void	Decrement reference count of bus driver.

Table 17: struct subsystem Programming Interface.

Programming Interface

The programming interface for bus drivers consists of registration and reference counting functions. Bus drivers should be registered in their startup routines, whether compiled statically or as a module. They should be unregistered in their module exit routines. Reference counts for bus drivers should be incremented before a bus driver is used and decremented once use of the driver is done.

There are two caveats of using bus drivers as they are currently implemented in the driver model . First, the de facto means of referencing a bus driver is via a pointer to its struct bus_type. This implies that the bus drivers must not declare the object as 'static' and must beexplicitly exported for modules to use. An alternative is to provide a helper that searches for and returns a bus with a specified name. Thisis a more desirable solution from an abstraction perspective, and will likely be added to the model.

Secondly, bus drivers contain no internal means of preventing their module to unload while their reference count is positive. This causes the referring object to access invalid memory if the module is unloaded. The proposed solution is to use a semaphore, like device drivers contain, that bus_unregister() waits on, and is only unlocked when the reference count reaches 0. This will be fixed inthe near future.

Driver Binding

Device drivers are discussed in a later section, but that does hinder the discussion of describing the process of binding devices to drivers. Binding happens whenever either a device or a driver is added to a bus. The end result is the same, though the means are slightly different. During attaching or detaching devices or drivers, the bus's rwsem is taken to serialize these operations.

When a device is added to the core, the internal function bus_add_device() is called. This inserts the device into the bus's list of devices, and calls device_attach(). This function iterates over the bus's list of drivers and tries to bind it to each one, until it succeeds. When the device is removed, bus_remove_device() is called, which detaches it from the driver by calling the driver's 'remove' method.

When a driver is added to the core, the internal function bus_add_driver() is

Name	*Return Type*	*Description*
bus_for_each_dev(struct bus_type * bus, void * data, int (*callback)(struct device * dev, void * data));	int	Iterator for accessing each device on a bus sequentially. It holds a read lock on the bus,while calling 'callback' for each device.
bus_for_each_drv(struct bus_type * bus, void * data, int (*callback)(struct device_driver * drv, void * data));	int	Identical to bus_for_each_dev(), but operates on a bus's registered drivers.

Table 18: struct bus_type Helper Functions.

called. This inserts the driver into the bus's list of drivers, and calls driver_attach(). This function iterates over the bus's list of devices, trying to attach to each that does not already have a driver. It does not stop at the first successful attachment, as there may be multiple devices that the driver may control. When a driver module is unloaded, bus_remove_driver() is called, which iterates over the list of devices that the driver is controlling. The driver's 'remove' method is called for each one.

During the attachment of either a device or a driver, the function bus_match() is called at each iteration. This calls the bus's 'match' method is called to compare the device ID of the device with the table of IDs that the driver supports. 'match' should return TRUE (1) if one of the driver's IDs matched the device's, or FALSE (0) if none of them did.

If 'match' returns TRUE, the device's 'driver' pointer is set to point to the driver, and the driver's 'probe' method is called. If 'probe' returns success (0), the device is inserted into the driver's list of devices that it supports.

Bus Helper Functions

Occasionally, it is useful to perform an operation on the entire space of devices or drivers that a bus supports. To aid in this, the driver model core exports a helper to do each.

bus_for_each_dev() iterates over each device that a bus object knows about. For each device, it calls the callback passed in, passing the current device pointer, and the data that was passed as a parameter to the function. bus_for_each_drv() behaves identically, though it operates on the bus's driver-space.

Each function checks the return value of the callback after calling it, and returns immediately if it is non-zero.

Each function increments the reference count of the object it's iterating on before it calls the callback, and releases it when it returns.

Each functions takes a read lock for the bus. This allows multiple iterations to be taking place at the same time, but prevents against a removal or addition happening while any iterations are in progress. These functions always increment the reference count of the object before calling the callback, and decrement

it after it returns. The function does not return whichobject caused the callback to return a non-zero result.

The callback must store this information if it needs to retain it. It must also increment the reference count of the object to guarantee the pointer remains valid until it can access it. The example below illustrates this point.

```
struct callback\_data {
       struct device * dev;
       char * id;
};

static int callback(struct device * dev, void * data)
{
        struct callback\_data * cd = (struct callback\_data *)data;
        if (!strcmp(dev->bus\_id,cd->id)) {
                cd->dev = get\_device(dev);
                return 1;
        }
        return 0;
}

static int caller(void)
{
        struct callback\_data data = {
                .id = "00:00.0",
        };

        /* find PCI device with ID 00:00.0  */
        if(bus\_for\_each\_dev(&pci\_bus\_type,&data,callback)) {
                struct device * dev = data.dev;
                /* fiddle with device */
                put\_device(dev);
        }
}
```

7 Device Drivers

A device driver is a body of code that implements one or more interfaces of a device class for a set of devices on a specific bus type. Drivers are specific to both their class and bus. Only in rare cases may drivers work on devices on two different bus types, and in those cases, it is because of a abstraction layer internal to the driver. Device drivers also contain a set of methods to perform bus-specific operations for their target. The driver model defines struct device_driver to describe device drivers.

Device drivers are optional and may be loaded as modules. They contain some statically allocated structures to describe the driver to its bus and class, and to maintain a list of devices bound to it. During startup, a driver registers an object with its bus. This object contains basic description information, the bus-specific methods it implements, and a table of device IDs supported by the driver. The bus driver inserts the driver into a list, and attempts to bind the

Name	*Return Type*	*Description*
name	char *	The name of the driver.
bus	struct bus_type	The bus the driver belongs to.
devclass	struct device_class	The class the driver belongs to.
unload_sem	struct semaphore	Semaphore to protect against a driver module being unloaded while the driver is still referenced.
kobj	struct kobject	Generic object information.
class_list	struct list_head	List entry in class's list of drivers.
devices	struct list_head	List of devices bound to driver.

Table 19: struct device_driver Data Fields.

driver to devices present on the bus. When a driver is bound to a device, it allocates a private structure, and usually allocates the class-specific object for the device. Traditionally, the device has registered the device with the class manually, though the driver model is working to move that functionality out of the drivers.

A driver is detached from a device if the driver module is unloaded. This involves unregistering the device from the class and freeing the private structure ithad allocated for the device.

Structural Overview

struct device_driver is designed to describe a device driver at a simple level. It consists of data elements that are independent of bus or class types, linkage information that is common to all buses and classes, and a set of methods designed to be called from driver model core code.

Currently, the generic driver structure is intended to be embedded in the bus-specific driverstructures that already exist today. The bus driver is then modified to forward the registration call to the driver model core when the driver registerswith its bus. This allows a gradual transition phase while the individual driversaremodified to use the generic fields.

To smooth this transition, the bus drivers also supply parameters for the generic structure on behalf of the bus-specific structures. This is true for the methods of struct device_driver. Many bus-specific driver objects have methods similar to those in the generic driver structure. A bus driver may implement methods for the generic structure that simply forward method calls to methods in the bus-specific structure. Drivers can become part of the driver model without having to modify every driver structure.

The bus drivers may also set the bus type for the driver, since it receives the intermediate call. However, the driver should specify its device class, since the class has no way of determining that.

Name	*Return Type*	*Description*
probe (struct device * dev)	int	Called to verify driver can be bound to device, and attach to device.
remove(struct device * dev)	int	Called to detach driver from device.
shutdown(struct device * dev)	void	Called during system shutdown sequence to quiesce devices.
suspend(struct device * dev, u32 state, u32 level)	int	Called during system suspend transition to put device to sleep.
resume(struct device * dev, u32 level)	int	Called during system resume transition to wake device back up.

Table 20: struct device_driver Methods.

The 'kobj' member contains genericobject meta data. This object belongs to the driver subsystem of the driver's bus type. It is registered with the subsystem when the driver registers with the driver model core. The 'class_list' field is a list entry that allows the drive to be inserted in its class's list of drivers. The 'devices' list is a list of devices that have been attached to the driver.

The 'unload_sem' member is used by the core to prevent a driver module from being unloaded while its reference count it still positive. This semaphore is initialized to a locked state, and only unlocked when a driver's reference count reaches 0. A driver may be unregistered at any time, even if the reference count is positive. Unfortunately, after a driver is unloaded, its module is removed unconditionally. If unregistering the driver did not remove the last reference, the process blocks waiting for the reference count to reach 0, and the semaphore to be released.

The methods of struct device_driver are operations that the driver model core calls when it needs to perform device-specific actions during an otherwise generic operation. As before, the bus may implement these methods for all registered drivers, and simply forward the calls to the bus-specific driver methods.

'probe' is called to determine whether a device can support a specific device. This is called during the driver binding operation, after the bus has verified that the device's ID matches a supported ID. This method is for the driver to verify that the hardware is in good, working condition and it can be successfully initialized.

This method is traditionally used to initialize the device and register it with the driver's class. Discussion has surfaced several times about separating the 'probe' method into 'probe' and 'start' (or similar). 'probe' would only verify that the hardware was operational, while 'add' would handle initialization and registration of the device. Some common registration functionality could also be moved out of the drivers and into the core, since it typically happens after operational verification and before hardware initialization. For now, 'probe' remains dual-purposed and monolithic.

Name	*Return Type*	*Description*
driver_register(struct device_driver *)	int	Register driver with core.
driver_unregister(struct device_driver *)	void	Unregister driver from core.
get_driver(struct device_driver *)	struct device *	Increment driver's reference count.
put_driver(struct device_driver *)	void	Decrement driver's reference count.

Table 21: struct device_driver Programming Interface.

'remove' is used to disengage the device from the driver. The driver should shut down the device. 'remove' is called during the driver unbinding operation, when either a device or a driver has been removed from the system.

'shutdown' is called during a reboot or shutdown cycle. It is intended to quiesce the device during a shut down or reboot transition. It is called only from device_shutdown(), during a system reboot or shutdown sequence.

'suspend' and 'resume' are called during system-wide and device-specific power state transitions, though only it is currently only used in the former. Power management is discussed in its own section, and should be consulted for the details of the parameters of these functions.

Programming Interface

Device drivers have a programming interface similar to the other driver model objects. They have register and unregister functions, as well as routines to adjust the reference count. The only behavioral difference is usage of the driver's 'unload_sem' field, as described above.

8 Device Classes

A device class describes a function that a device performs, regardless of the bus on which a particular device resides. Examples of device classes are:

- Audio output devices.
- Network devices.
- Disks.
- Input devices.

A device class is characterized by a set of interfaces that allow user processes to communicate with devices of their type. An interface defines a protocol for communicating with those devices, which is usually characterized in a user space device node. A class may contain multiple interfaces for communicating with

Name	*Return Type*	*Description*
name	char *	Name of device class.
devnum	u32	Value for enumerating devices registered with class.
subsys	struct subsystem	Collection of subordinate interfaces.
devsubsys	struct subsystem	Collection of devices registered with class.
drvsubsys	struct subsystem	Collection of drivers registered with class.
devices	struct list_head	List of devices registered with class.
drivers	struct list_head	List of drivers registered with class.

Table 22: struct device_class Data Fields.

Name	*Return Type*	*Description*
add_device(struct device *)	int	Method called to register device with class. Called after device has been bound to driver belonging to class.
remove_device(struct device *)	void	Called to unregister device from class. Called while detaching device from driver that belongs to class.
hotplug(struct device *dev, char **envp, int num_envp, char *buffer, int buffer_size)	int	Called before /sbin/hotplug is called when device is registered with class. This is opportunity for class to fill in and format parameters to /sbin/hotplug.

Table 23: struct device_class Methods.

devices, though the devices and drivers of that class may not support all the interfaces of the class. Device interfaces are described in the next section.

A driver for a device class maintains lists of device and drivers that belong to that class. They are optional bodies of code that may be compiled as a module. Its data structures are statically allocated. A device class defines object types to describe registered devices.These objects define the devices in the context of the class only, since classes are independent of the registered devices' bus type.

The driver model defines struct device_class to represent a device class driver. The driver model does not explicitly represent the per-device objects that device class drivers operate on, though they are expected to use the class_data member of struct device to store that object.

Structural Definition

Struct device_class most closely resembles struct bus_type. Indeed, many of the fields of struct device_class server a similar purpose as those in struct bus_type.The subordinate subsystem 'devsubsys' and the 'devices' list manage the list of devices registered with the class. Like with buses, these structures must exist in parallel, since a kobject may not belong to more than one subsystem at a time. The same is true of the 'drvsubsys' and 'drivers' list with regard to drivers registered with the classes.

The 'subsys' member manages the list of interfaces registered with the class. As interfaces are registered, they are inserted into the subsystem's list. This member is also used to register the class with the kobject hierarchy, as a member of the global class subsystem.

The 'devnum' field is an enumerator for devices that are attached to the class. A device's 'class_num' field is set to this value when a device is registered with a class. The 'devnum' field is incrementedafter each time a device is enumerated.

Device classes contain three methods that are called by the driver model core during the process of adding and removing devices from a class. add_device() is called when a device is registered with the device class. A device is registered with a device class immediately after the device is bound to a driver. It is added to the class that the driver belongs to. remove_device() is called when a device is unregistered from a device class. This happens when a device is being detached from its driver.

The 'hotplug' method is called immediately after a device is added or removed from a device class. This allows the class to define additional environment variables to set for the hotplug agent when the driver model core executes it.

Programming Interface

Device classes have a similar interface to other driver model objects. It offers calls to register and unregister device classes, as well asfunction to adjust the reference count. Device class drivers should statically allocate struct device_class objects. They should initialize the 'name' field, and the pointers of the methods they support. The object should be registered in the class's initialization function, and unregistered in the class's tear down function, if it is compiled as a module.

9 Device Interfaces

A device class represents the functional type of a device, but it is characterized by the interfaces to communicate with that type of device. A device class interface defines a set of semantics to communicate with a device of acertain type. These are most often in the form of device nodes visible to user space, though they could also beregular files in a filesystem exported by the kernel.

There may be multiple interfaces per device type, and not all devices or drivers of a class may support all the interfaces of that class. For example, the

Name	*Return Type*	*Description*
devclass_register(struct device_class *)	int	Register class with core.
devclass_unregister(struct device_class *)	void	Unregister class with core.
get_devclass(struct device_class *)	struct device_class *	Increment class's reference count.
put_devclass(struct device_class *)	void	Decrement class's reference count.

Table 24: struct device_class Programming Interface.

Name	*Return Type*	*Description*
name	char *	Name of interface (must be unique only for class).
devclass	struct device_class *	Class interface belongs to.
subsys	struct subsystem	Collection of subordinate objects.
devnum	u32	Enumerator for registered devices.

Table 25: struct device_interface Data Fields.

input device class describes devices that are capable of generating input for the kernel. A device and driver may support multiple ways of communicating with it. For example, a touch screen may be accessed as a mouse device, as well as a touch screen device. However, a mouse device may be accessed using only the mouse interface, and not the touchscreen interface.

The driver model defines struct device_interface to describe device interfaces. They are simple structuresthat may be dynamically registered and unregistered with device classes. A class's interfaces are referenced when a device is added to or removed from the class. When a device is added, the driver model attempts to add the device to every interface registered with the class.

A device may not support all of the interfaces registered with the device, so the driver model defines a separate object type to express a device's membership with an interface. An instance of this object is added to a device's list. When a device is removed from a class, it is removed from the interfaces to which it had been added by iterating over this list, rather than the list of the class's interfaces.

Driver Model Representation

struct device_interface is similar to struct device_class, though simpler. Its member fields and methods are described below.

An interface's 'subsys' member contains generic object meta data and isregistered as subordinate of thesubsystem of the interface's device class. It is also used to contain the device-specific objects allocated by the interface.

The 'devnum' member is used to enumerate devices when they are registered

Name	*Return Type*	*Description*
add_device(struct device *)	int	Called to register device with interface, after device has been registered with class.
remove_device(struct intf_data *)	int	Called to unregister device from interface, before device has been unregistered from class.

Table 26: struct device_interface Methods.

Name	*Return Type*	*Description*
interface_register(struct device_interface *)	int	Register interface with class.
interface_unregister(struct device_interface *)	void	Unregister interface from class.

Table 27: struct device_interface Programming Interface.

with the interface. When an interface attaches the device-specific data to the interface,it is assigned the value of 'devnum', which is then incremented.

Programming Interface

Interface objects should be statically allocated and registered on startup. They should initialize the 'name' and 'devclass' field, as well as the methods they support. Interfaces support only a registration interface. There is no explicit mechanism to perform reference counting on them, though one could do so directly on the embedded struct subsystem.

Interface Data

The driver model defines an object, struct intf_data, to describe a device-specific object for aninterface. This objectcan beregistered as an object of the interface's embedded subsystem and added to the 'intf_list' member of struct device. This allows interfaces to maintain an accurate list of registered devices, and for devices to maintain an accurate list of interface membership.

This is necessary since devices may not belong to all of the interfaces of a device class, and an interface may not be valid for all devices registered with a device class. When a device isremoved from a device class, the driver model core can iterate over the device's list of intf_data objects to reference its containing interfaces. And, when an interface is removed from a class, the driver model core can iterate over the interface's subordinate kobjects - since they are embedded in struct intf_data - to reference the attached devices.

struct intf_data may be embedded in a more complex device-specific data structure for the interface, or may be allocated separately. It should be allocated

Name	*Return Type*	*Description*
intf	struct device_interface	Interface data belongs to.
dev	struct device *	Device being registered with interface.
intf_num	u32	Interface-enumerated value of this object.
dev_entry	struct list_head	List entry in device's list of interface objects.
kobj	struct kobject	Generic object data.

Table 28: struct intf_data Data Members.

Name	*Return Type*	*Description*
interface_add_data(struct intf_data *)	int	Attach data object to interface and device. This should usually be called during the interface's add_device() method once it's determined the device can support it.

Table 29: struct intf_data Programming Interface.

during the core's call to the interface's add_device method. This method should then call interface_add_data() to attach the data to the deivce and the interface. The interface should initialize the 'intf' and 'dev' members of the intf_data object before calling interface_add_data().

When a device is removed from an interface, the internal function interface_remove_data() is called immediately before the interface's remove_device() method. Thisdetaches it from the deviceand the interface, so that the interface may free the structure in their remove_device() method.

10 Platform and System Devices

The discussion of the driver model thus far has focused on peripheral expansion buses that tend to share many characteristics in their handling of devices and drivers, despite having radically different physical attributes. The model works well for them because it consolidates often-replicated object management code present in each of those bus drivers. This covers a vast majority of devices that the kernel supports.

However, the driver model must make exceptions for two classes of devices: system-level devices, legacy I/O devices, and host bridges. Legacy devices and host bridges are grouped into a common category - 'platform devices', since they are an integral part of the platform and the physical makeup of the system board.

System Devices

System-level devices are devices that are integral to the routine operation of the system. This includes devices such as processors, interrupt controllers, and system timers. System devices do not follow normal read/write semantics. Because of this, they are not typically regarded as I/O devices, and are not represented in any standard way.

They do have an internal representation, since the kernel does communicate with them, and does expose a means to exert control over some attributes of some system devices to users. They are also relevant in topological representations. System power management routines must suspend and resume system devices, as well as normal I/O devices. And, it is useful to define affinities to instances of system devices in systems where there are multiple instances of the same type of system device.

These features can happen in architecture-specific code. But, the driver model's generic representation of devices provides an opportunity to consolidate, at least partially, the representations into architecture-independent ones.

Problems

This representation raises a few exceptions in the driver model. For one, there is no common controlling bus that system devices reside on. Computers always have at least one system-level bus on which the CPU and memory usually reside. This device is always present, and not probed for like buses of other types. To accommodate for this, a struct bus_type object is allocated for the system bus and registered on startup. This 'pseudo-bus' is intended to represent the controlling bus of all system devices.

Secondly, system devices are dynamically discovered by bus probe routines, like devices on other bus types are. There is no common way to communicate with more than one type of system device, so this is quite impossible. Devices are discovered via very specific operations in driver-like code.

The term 'driver-like' is used because system devices typically do not have device drivers like most peripheral devices do. The purpose of most device drivers is to implement device-specific support for a programming interface designed for a certain class of devices. Most system-level devices are in a class of their own, so there is no need to register their existence, or abstract their specifics. But, there are drivers that do initialization of system-level devices, and export them to other parts of the kernel.

This makes the discovery of system devices dependent on the presence of the device's driver. In other peripheral buses, device discovery and driver registration are two mutually exclusive operations. This does not completely break the model, but it causes system device infrastructure to make obtuse calls to the driver model core during registration.

System devices are expected to register a device class object first, then a device driver for the device type, setting its bus to be the system pseudo-bus object, and its class to be the device class that was just registered. Then,

Name	*Return Type*	*Description*
id	u32	Instance number of this system root.
dev	struct device	Generic device information for system root.
sysdev	struct device	Statically-allocated virtual system bridge for this system root.

Table 30: struct subsystem Programming Interface.

Name	*Return Type*	*Description*
name	char *	Canonical name of system device.
id	u32	Enumerated instance of system device.
root	struct sys_root *	System root device exists on.
dev	struct device	Generic device information.

Table 31: struct subsystem Programming Interface.

the devices should actually be registered, with their bus set to the system bus object, and their driver set to the driver just registered. This is not the most elegant solution, but it allows system devices to have complete coverage within the driver model.

Representation

System devices are described in the following way:

struct sys_root is designed to accommodate systems that are composed of multiple system boards, though are regarded as having a contiguous topology. NUMAQ systems are an example of a platform with this feature that are composed of four system boards bridged together with a custom controller to maintain cache coherency.

To accurately represent the topology of the system, each system board is represented by a struct sys_root. On all systems, there is an implicit sys_root present. On NUMAQ systems, additional system roots may be registered to represent the different system boards. Each of these subordinate roots contain a struct device to represent a logical bridge to the system bus on that board.

When system devices are discovered, their sys_root pointer may be set to be the root under which they physically reside. They will be added as children of the system 'bridge' of the root under which they reside. If their root is not set, they will be added under the default root. System devices are always added as members of the system bus type.

Both struct sys_root and sys_device contain 'id' fields, and struct sys_device contains a 'name' field. Device discovery routines can use these fields to name

Name	*Return Type*	*Description*
name	char *	Canonical name of platform device.
id	u32	Enumerated instance of platform device.
dev	struct device	Generic device information.

Table 32: struct subsystem Programming Interface.

enumerate the devices that are being registered for easy identification purposes later, without having to reference the embedded struct device.

Platform Devices

Platform devices are the set of legacy I/O devices and host bridges to peripheral buses. These sets of devices share the characteristic that they are devices on the system board itself; they are not expansion devices. They also share the characteristic with system devices that they are not necessarily part of a common controlling bus. Like system devices, their discovery is dependent on the presence of a driver for them. Although, many modern firmwares are attempting to change this, as will be described later.

To cope with platform devices, the driver model created a pseudo bus to pose as the bus driver for platform devices. However, there is no common parent for all platform devices. Legacy devices on modern systems reside on an ISA bus, a subordinate of the PCI-ISA bridge on the PCI bus. It is important to accurately describe the system topology, so the platform device model allows for registrants to encode this in the device's parent.

The driver model has also created a structure very similar to struct sys_device to describe platform devices.

The 'name' and 'id' are intended to be set by the discoverer of the devices to allow for easy identification and comparison, without having to reference or parse the embedded device's fields.

Discovery

Platform device discovery has traditionally occurred when a driver loaded and probed hard-coded I/O ports to test for existence. This can cause problems, though, when a driver is running on a platform where the ports to probe for existence are different. It doesn't even have to be different architectures, only different revisions of the same platform. Probing undefined I/O ports is dangerous and cause very unpredictable behavior. It can also be a very slow process, and significantly delay the boot process.

To cope with these problems, the drivers can be modified to use different I/O ports on different platforms, but it often convolutes the code.

Firmware

Many modern firmware implementations have made attempts to solve this problem by defining tables to describe the devices that are attached to the system board, since they are known when the firmware is built. The kernel can read these tables to know what type of devices are attached where, independent of any driver being loaded.

This creates a need to do dynamic driver binding for these devices. The platform bus is designed to handle this. Theoretically, a firmware driver could parse the firmware tables, and add platform devices for the devices that they describe. The drivers for these devices can be registered with the platform bus, and bound to the devices registered with it.

However, in order to use this infrastructure, drivers for platform devices must be modified to handle this. They must first register with the platform bus, and not probe for devices when they are loaded. Legacy probing mechanisms may still exist as a fall back, but they must be separated from the core initialization routines of these drivers. It may also be beneficial to make these parts of the drivers optional.

This feature also depends on the firmware drivers and device drivers using the same name to represent a type of device. Each firmware encodes a different way to describe each type of device, like a different Plug N' Play ID for each. Instead of modifying the common platform layer to know about every firmware driver's method of encoding each device type, firmware drivers are expected to map their local device descriptors into a common format.

For example, instead of naming a Host-PCI bridge after its PnP ID, "PNP0a03", it would name the device "pci", which is equally meaningful to all code, as well as someone reading the code. The PCI bus driver could then load and register a driver with name "pci". The platform bus would use these two names in its 'match' method to match the device with the driver, and call the PCI driver's probe method.

Work in this area is largely experimental. Not much quantifiable progress has been made, besides previous proof-of-concept implementations. More work is expected to take place in this area in the near future.